Java-Semantik mittels guarded commands

Stefan Schimanski, Frank Stamm

Institut für Informatik, Technische Universität Clausthal,
Julius-Albert-Strasse 4, 38678 Clausthal-Zellerfeld
{Stefan.Schimanski, Frank.Stamm}@informatik.tu-clausthal.de

Zusammenfassung

Der vorliegende Bericht beschreibt die Definition einer axiomatischen Semantik für Java-Programme. Dazu wird eine auf den *guarded commands* von Dijkstra basierende Spezifikationssprache mit formaler Semantik eingeführt, die die Darstellung wesentlicher Sprachkonstrukte der Java-Programmiersprache ermöglicht. Die hierfür notwendige Übersetzung von Java in die *guarded commands* wird dargestellt. Die Transformation eines vollständigen Beispielprogramms schließt den Bericht ab.

1 Einleitung

Die Definition einer formalen Semantik für eine moderne Programmiersprache wie Java ist im Allgemeinen eine fast unlösbare Aufgabe. Insbesondere, wenn sämtliche zur Verfügung stehenden Modellierungsmöglichkeiten in unbeschränktem Maße verwendet werden, stößt die Sematikdefinition an unüberwindbare Grenzen [1].

Für den Nachweis von wünschenswerten Eigenschaften eines Java-Programms ist es notwendig, eine Semantik für Java zu definieren, mit der es möglich ist, Beweisverpflichtungen in eine möglichst einfache Logik zu transformieren. Dies erlaubt dann insbesondere den Einsatz von (semi-)automatischen Beweisassistenten.

Viele der veröffentlichten Semantikansätze für Java [2] erfüllen diese Forderung nicht. Aus diesem Grund hat es sich als günstig erwiesen, zur Semantikdefinition Programme zunächst in eine einfachere Zwischensprache zu transformieren, für die eine Semantikdefinition einfacher zu realisieren ist [3].

Inspiriert durch [4] wird hier eine Übersetzung von Java-Programmen in eine an [5] angelehnte zustandsbasierte Spezifikationssprache beschrieben. Dazu werden zunächst im folgenden Abschnitt 2 die Grundlagen der axiomatischen Semantikdefinition nach Dijkstra bereitgestellt. In Abschnitt 3 wird dann die, zur formalen Beschreibung von Java-Programmen, verwendete Spezifikationssprache eingeführt. Abschnitt 4 beschreibt detailliert die Übersetzung wesentlicher Programmkonstrukte von Java in die definierte Spezifikationssprache. Diese Übersetzung wird in Abschnitt 5 anhand einer Implementierung des Algorithmus von Euklid betrachtet. Der letzte Abschnitt 6 fasst

die gewonnenen Erkenntnisse zusammen, und gibt einen Ausblick auf die durch die Übersetzung ermöglichten Arbeiten.

2 Grundlagen und Begriffe

In diesem Abschnitt werden einige Grundlagen und Begriffe eingeführt, auf die im weiteren Verlauf der Arbeit Bezug genommen wird. Es wird insbesondere auf das Zustandskonzept und die Theorie der Prädikattransformer eingegangen.

2.1 Zustandskonzept

Dem hier vorgestellten Ansatz zur Induzierung einer Semantik in Java-Programmen liegt die *axiomatische Programmsemantik* nach Dijkstra durch die Prädikattransformer $wlp(S)$ und $wp(S)$ zugrunde. Ausgangspunkt für die Entwicklung der axiomatischen Methode im Sinne von Dijkstra für die Semantik imperativer Sprachen ist die relationale Semantik.

Einem beliebigen Programmstück S kann, wenn Zwischenzustände beiseite gelassen werden, stets eine Menge von Zustandsübergängen (σ, τ) zugeordnet werden. Dabei wird o. B. d. A. davon ausgegangen, dass σ und τ auf derselben Variablenmenge definiert sind. Zur Modellierung von Nichtterminierung im Anfangszustand σ wird üblicherweise für τ ein zusätzlicher Wert ∞ verwendet.

Definition 1 Ein *Zustandsraum* ist eine endliche Menge X von Variablen, wobei jeder Zustandsvariablen $x \in X$ ihre Sorte $\sharp x$ zugeordnet wird. Eine Variablenbelegung $\sigma : X \to \bigcup_{x \in X} A_{\sharp x}$ mit $\sigma(x) \in A_{\sharp x}$ für Trägermengen $A_{\sharp x}$ zu den Sorten $\sharp x$ heißt *Zustand*. Es sei Σ die Menge aller Zustände über X und $\infty \notin \Sigma$. Die *Zustandsübergangsmenge* ist dann eine Teilmenge $\Delta(S) \subseteq \Sigma \times (\Sigma \cup \{\infty\})$. □

Für gegebenes S liegt nun die Hauptschwierigkeit in der Bestimmung von $\Delta(S)$. Prinzipiell könnte jeder endlichen Zustandsfolge $\sigma_0 \to \cdots \to \sigma_n$ ein Übergang (σ_0, σ_n) und jeder unendlichen Zustandsfolge $\sigma_0 \to \sigma_1 \to \cdots$ ein Übergang $(\sigma_0, \infty) \in \Delta(S)$ zugeordnet werden. Dieses Objekt ist für den Nachweis von Programmeigenschaften allerdings wenig geeignet, da die Zustandsfolgen nicht a priori bestimmbar sind. Aus diesem Grund werden in der Programmsemantik nach Dijkstra Teilmengen von Σ durch prädikatenlogische Formeln φ mit freien Variablen in X ausgedrückt. Dazu wird eine mehrsortige prädikatenlogische Sprache $\mathcal{L}$ als syntaktische Basis für die weiteren Ausführungen benötigt.

Definition 2 Ein Quadrupel $\mathcal{L} = (S, \mathfrak{F}, \mathfrak{P}, \{V_s\}_{s \in S})$, bestehend aus

- einer höchstens abzählbaren Menge S von *Sorten*,
- einer Familie $\mathfrak{F}$ von Mengen $\mathfrak{F}_k$, in denen jedem k-stelligen Funktionssymbol $f \in \mathfrak{F}_k$ ein Definitionsbereich $s_1 \ldots s_k$ und ein Bildbereich s mit $s, s_i \in S$ $(1 \leq i \leq k)$ zugeordnet wird,

- einer weiteren Familie $\mathfrak{P}$ von Mengen $\mathfrak{P}_k$, in denen jedem $k-$stelligen Prädikatsymbol $P \in \mathfrak{P}_k$ ein Definitionsbereich $s_1 \ldots s_k$ mit den Sorten $s_i \in S$ $(1 \leq i \leq k)$ zugewiesen wird, und schließlich
- einer abzählbaren Mengen V_s von *Variablen der Sorte* $s \in S$,

heißt eine *mehrsortige prädikatenlogische Sprache*. □

Die Menge der *Terme* $\mathbb{T}$, die Menge der *atomaren Formeln* $\mathbb{F}_0$ und die Menge der *Formeln* $\mathbb{F}$ der prädikatenlogischen Sprache $\mathcal{L}$ werden wie üblich, siehe z. B. [6], gebildet. Für die folgende Theorie der Prädikattransformer ist es allerdings notwendig unendliche Konjunktionen und Disjunktionen zuzulassen. In Formeln dürfen aber nur endlich viele freie Variablen und endlich viele Quantoren auftreten. In der Literatur wird diese Logik auch mit $\mathcal{L}^{\omega}_{\omega\infty}$ bezeichnet.

Zu beachten ist, dass Variablen in Formeln sowohl frei als auch gebunden vorkommen können. Die Menge der freien Variablen einer Formel φ wird im Folgenden mit $fr(\varphi)$ bezeichnet.

Freie Variablen in einer Formel können durch beliebige Terme ersetzt werden. Sei $\varphi \in \mathbb{F}$ eine Formel und $x \in fr(\varphi)$ eine hierin nur frei vorkommende Variable. Ist nun $t \in \mathbb{T}$ ein Term, bei dem keine Variable aus t gebunden in φ vorkommt, so bezeichnet $\{x/t\}.\varphi$ die Formel, die durch Ersetzung von x durch t in φ entsteht. Diese Operation wird auch als *Substitution* bezeichnet.

Zur Festlegung einer Semantik der prädikatenlogischen Sprache $\mathcal{L}$ muss zunächst den Funktions- und Prädikatsymbolen eine Funktion bzw. Relation zugeordnet werden. Es bezeichne $\mathbb{B} = \{\mathbf{W}, \mathbf{F}\}$ die Menge der *Wahrheitswerte*. Auf $\mathbb{B}$ können dann Operationen eingeführt werden, die die umgangssprachliche Verwendung von Negation, Konjunktion oder Disjunktion formalisieren.

Definition 3 Es bezeichne $\mathcal{L} = (S, \mathfrak{F}, \mathfrak{P}, \{V_s\}_{s\in S})$ eine mehrsortige prädikatenlogische Sprache. Eine *Struktur* für $\mathcal{L}$ ist dann ein Paar $(\{D_s\}_{s\in S}, \omega)$ mit einer sortenindizierten Familie von Mengen D_s, die *semantische Bereiche* der Sorte $s \in S$ heißen, und einer Abbildung ω mit Definitionsbereich $\bigcup_{i\in\mathbb{N}} \mathfrak{F}_i \cup \bigcup_{i\in\mathbb{N}} \mathfrak{P}_i$. Diese ordnet jedem n-stelligen Funktionssymbol $f \in \mathfrak{F}_n$ mit Definitionsbereich $s_1 \ldots s_n$ und Bildbereich s eine Abbildung $\omega(f) : D_{s_1} \times \cdots \times D_{s_n} \to D_s$ und jedem n-stelligen Prädikatsymbol $P \in \mathfrak{P}_n$ mit Definitionsbereich $s_1 \ldots s_n$ eine Abbildung $\omega(P) : D_{s_1} \times \cdots \times D_{s_n} \to \mathbb{B}$ zu. □

Zur Interpretation von Formeln müssen die in ihnen vorkommenden Variablen durch Elemente des passenden semantischen Bereiches belegt werden.

Definition 4 Sei $St = (\{D_s\}_{s\in S}, \omega)$ eine Struktur für die mehrsortige prädikatenlogische Sprache $\mathcal{L} = (S, \mathfrak{F}, \mathfrak{P}, \{V_s\}_{s\in S})$. Eine *Variablenbelegung* ist eine Familie $\{\sigma_s\}_{s\in S}$ von Abbildungen $\sigma_s : V_s \to D_s$. Eine *Interpretation* von $\mathcal{L}$ ist dann das Paar $I = (St, \{\sigma_s\}_{s\in S})$. □

Die Fortsetzung der Interpretation auf Terme und Formeln mittels $\omega^I_{\mathbb{T}}$ bzw. $\omega^I_{\mathbb{F}}$ erfolgt dann wiederum wie üblich, siehe z.B. [6].

2.2 Grundlegende Theorie der Prädikattransformer

Für die weiteren Betrachtungen wird die verwendete mehrsortige prädikatenlogische Sprache $\mathcal{L} = (S, \mathfrak{F}, \mathfrak{P}, \{V_s\}_{s \in S})$ zunächst fixiert.

Dazu sei S die Menge der Sorten der verwendeten Programmiersprache und $\mathfrak{F}_i$ die Menge der definierten *seiteneffektfreien* i-stelligen Operatoren. Als nullstelliges Prädikatsymbole werden $True$ und $False$ verwendet, $\mathfrak{P}_2$ enthalte die von der Programmiersprache unterstützten zweistelligen Prädikate zu jeder Sorte $s \in S$. Alle weiteren $\mathfrak{P}_i$ seien leer. Die Variablenmengen V_s enthalten jeweils eine Kopie der zugelassenen Variablenbezeichner. Auf die zugehörige Sorte einer Variablen wird später kontextbezogen mittels *Typisierung* geschlossen.
Die benötigte Struktur $St = (\{A_s\}_{s \in S}, \omega)$ kann ebenfalls fixiert werden. Dazu seien A_s die vorausgesetzten Trägermengen, $\omega(f)$ die $f : s_1 \dots s_n \to s$ entsprechende Abbildung $A_{s_1} \times \cdots \times A_{s_n} \to A_s$, $\omega(True) = \mathbf{W}$, $\omega(False) = \mathbf{F}$ und $\omega(P)$ die $P : s_1 \dots s_n \to \mathbb{B}$ entsprechende Abbildung $A_{s_1} \times \cdots \times A_{s_n} \to \mathbb{B}$.

Es wird gefordert, dass diese Struktur die so genannte *Abschlusseigenschaft* erfüllt, d.h. dass zu jedem $d \in A_s$ ein variablenfreier Term $t \in \mathbb{T}_s$ mit $\omega^I_{\mathbb{T}_s}(t) = d$ existiert. Diese Forderung stellt keine Einschränkung dar, da jede semantische Konstante die nicht dieser Eigenschaft genügt, mit einer entsprechenden Erweiterung der Interpretation als syntaktische Konstante zu der Termmenge hinzugefügt werden kann.

Da eine fixierte Struktur vorliegt, ist die Interpretation von Termen und Formeln nunmehr nur noch von der Belegung der Variablen abhängig. Die Variablen werden dabei nicht als unüberschaubare Zustandsmengen betrachtet, sondern als einfacher handhabbare Zustandsformeln in $\mathbb{F}$.

Definition 5 Eine *Zustandsformel* über einem Zustandsraum X ist eine Formel $\varphi \in \mathbb{F}$ mit freien Variablen in X. □

Jeder Formel $\varphi \in \mathbb{F}$ kann eine Zustandsmenge $\Sigma_\varphi = \{\sigma \in \Sigma \mid \omega^\sigma(\varphi) = \mathbf{W}\}$ zugeordnet werden, die durch die Formel φ charakterisiert wird. Es gilt auch die Umkehrung, d. h. zu einer Menge von Zuständen $\mathcal{Z} \subseteq \Sigma$ existiert eine Formel $\varphi_{\mathcal{Z}}$ mit $\Sigma_{\varphi_{\mathcal{Z}}} = \mathcal{Z}$.

Es stellt sich nun die Frage, von welcher Menge $\mathcal{Z}_0$ aus Zustände in einer gegebenen Menge $\mathcal{Z} \subseteq \Sigma$ erreicht werden können. Bezüglich der eben eingeführten Begriffe soll der charakterisierenden Formel $\varphi_{\mathcal{Z}}$ die charakterisierende Formel $\varphi_{\mathcal{Z}_0}$ zugeordnet werden.

Definition 6 Sei S ein Programmstück, gegeben durch die Zustandsübergangs-menge $\Delta(S)$ über dem Zustandsraum X.

- Die Abbildung $wp(S)$ ordnet einer Zustandsformel φ eine charakterisierende Formel der Zustandsmenge

  $$\{\sigma \in \Sigma \mid \omega^\tau(\varphi) = \mathbf{W} \text{ und } \tau \neq \infty \text{ für alle } \tau \in \Sigma \cup \{\infty\} \text{ mit } (\sigma, \tau) \in \Delta(S)\}$$

  zu. Die Formel $wp(S)(\varphi)$ ist die *schwächste Vorbedingung* von φ unter S.

- Die Abbildung $wlp(S)$ ordnet einer Zustandsformel φ eine charakterisierende Formel der Zustandsmenge

$$\{\sigma \in \Sigma \mid \omega^{\tau}(\varphi) = \mathbf{W} \text{ für alle } \tau \in \Sigma \text{ mit } (\sigma, \tau) \in \Delta(S)\}$$

  zu. Die Formel $wlp(S)(\varphi)$ ist die *schwächste Vorbedingung* von φ unter S *modulo Terminierung*. □

Informell haben die Prädikattransformer die folgende Bedeutung:

- $wp(S)(\varphi)$ charakterisiert gerade die Menge der Anfangszustände, für die jede Ausführung von S terminiert und zu einem Endzustand führt, der φ erfüllt.
- $wlp(S)(\varphi)$ ist gerade in solchen Anfangszuständen σ erfüllt, für die jede terminierende Ausführung von S zu einem Endzustand τ führt, in dem die gegebene Nachbedingung φ erfüllt ist.

Als zentrales Ergebnis der axiomatischen Semantik konnte gezeigt werden [7], dass die Relation zwischen den Anfangs- und Endzuständen und die Menge der nicht terminierenden Berechnungen durch die beiden Prädikattransformer $wlp(S)$ und $wp(S)$ vollständig beschrieben wird. Das heißt, diese beiden Prädikattransformer reichen aus, um die Semantik eines Programmstücks S eindeutig zu definieren.

3 Zustandsbasierte Spezifikationssprache

In diesem Abschnitt wird zunächst die zur formalen Beschreibung von Java-Programmen eingesetzte Spezifikationssprache betrachtet und eingeführt. Es handelt sich dabei um eine an die Spezifikationssprache aus [5] angelehnte *zustandsbasierte Spezifikationssprache*.

In dieser Spezifikationssprache wird zunächst grundsätzlich zwischen der statischen und der dynamischen Spezifikation unterschieden.

3.1 Statische Spezifikation

Bei der *statischen Spezifikation* handelt es sich um die Spezifikation eines Programms S, welche unter Vernachlässigung aller Ein- und Ausgaben die Werte von Programm- bzw. Zustandsvariablen beschreibt. Dazu sei $X = \{x_1, \ldots, x_n\}$ ein Zustandsraum. Eine statische Spezifikation für ein Programm S über X muss zunächst die Mengen möglicher Werte der Variablen aus X beschreiben. Für diese Beschreibung stehen die folgenden vordefinierten Mengen und Mengenkonstruktoren zur Verfügung:

- $BOOL$ enthält die Wahrheitswerte $True$ und $False$.
- $CHAR$ enthält alle Zeichen eines Alphabets $\mathcal{A}$. Die Zeichen sind dabei immer in einfache Anführungszeichen eingeschlossen.

- $STRING$ enthält alle aus den Zeichen des Alphabets $\mathcal{A}$ bildbaren Zeichenketten. Die Zeichenketten sind dabei immer in Anführungszeichen eingeschlossen.
- NAT, INT und $FLOAT$ enthalten die natürlichen, die ganzen bzw. die Gleitkommazahlen.
- *ID* enthält die abstrakten Objektidentifikatoren, über deren Bildungsregeln nichts vorausgesetzt wird. Zusätzlich enthält ID den speziellen Wert $Null$.

Über diese Basismengen können mittels *Abbildungen* und *Mengengleichungen* neue Mengen definiert werden. Eine Abbildung $(X \mapsto Y)$ zwischen zwei Mengen X und Y ist wie üblich durch die Menge der rechtseindeutigen Paare des kartesischen Produktes von X und Y definiert. Eine Mengengleichung bestimmt aus einer Menge die Elemente, die eine gegebene Formel erfüllen.

Eine *Mengenspezifikation* ist dann durch einen Ausdruck der Form

`Set` <set name> '=' <set expression>

gegeben.

Beispiel 1 Die Ausdrücke

$$\texttt{Set bool} = \text{BOOL} \text{ und}$$
$$\texttt{Set byte} = \{I \in INT \mid -2^7 \leq I \leq 2^7 - 1\}$$

stellen gültige Mengenspezifikationen in der Spezifikationssprache dar. □

Die Beschreibung von Daten in Form der definierten Mengen erfolgt durch Zustandsvariablen des entsprechenden Typs. In der Spezifikation wird dazu die Notation

`Variable` <variable name> '$\in$' <set expression> = <term>

verwendet. Der Term muss dabei einen Wert der zugehörigen Menge darstellen. Nach Definition 1 führt diese Verwendung von Variablen zu Zustandsräumen.

Beispiel 2 Die folgende Spezifikation definiert eine Variable x des Typs `byte` und initialisiert diese mit dem Standardwert:

$$\texttt{Variable}\, x \in \texttt{byte} \;\; = 0 \quad .$$

□

In einer Spezifikation können Elemente mittels benannter *Konstanten* mehrfach referenziert werden. Solche Konstanten können durch

`Constant` <constant name> '$\in$' <set expression> '=' <term>

definiert werden, wobei der definierende Term nur Variablen enthalten darf, wenn diese in der Spezifikation vorher definiert und initialisiert worden sind.

Beispiel 3 Durch die Spezifikation

$$\texttt{Constant}\ \text{maxbyte} \in NAT = 2^7 - 1$$
$$\texttt{Constant}\ \text{minbyte} \in NAT = -2^7$$

erfolgen gültige Konstantendefinitionen. □

3.2 Dynamische Spezifikation

Im vorhergehenden Abschnitt wurde die Spezifikation von Zustandsräumen betrachtet. Zustände sind Instanzen des Zustandsraumes und ermöglichen das statische Beschreiben der aktuellen Eigenschaften eines Programms.

Zur Beschreibung vom dynamischen Verhalten werden Mechanismen zum *Zustandsübergang* und zur *Initialisierung von Zustandsräumen* benötigt. Diese Mechanismen liefern die von Dijkstra [8] eingeführten und von Nelson [9] erweiterten *guarded commands* . Dabei handelt es sich bei den *guarded commands* um eine Sprache zur Spezifikation von elementaren Anweisungen, d. h. von Zustandsübergängen. Die Grundbausteine dieser Sprache bilden die *Zuweisung* von Werten an Zustandsvariablen, die effektlose Anweisung *Skip* und die totale Endlosschleife *Loop*.

Über diesen Grundbausteinen können die Konstruktoren *Sequenz, Auswahl, bedingte Auswahl, Vorbedingung, Projektion* und die *Rekursion* verwendet werden.
Die hier definierte *Variante* der *guarded commands* umfasst neben den ursprünglich definierten Konstrukten eine Erweiterung zur Beschreibung von *Operationsanwendungen* und die Möglichkeit der *operationalen Spezifikation* [5].

Definition 7 Sei X ein Zustandsraum. Die Menge der *guarded commands* auf X ist die kleinste Menge $\mathcal{G}(X)$, für die gilt:

- *Skip* und *Loop* liegen in $\mathcal{G}(X)$.
- Für eine Zustandsvariable $x \in X$ und einen Term $t \in \mathbb{T}$ mit $fr(t) \subseteq X$ liegt die *Zuweisung* $x := t$ auch in $\mathcal{G}(X)$.
- Für $S \in \mathcal{G}(X)$ und $T \in \mathcal{G}(Y)$ mit einem Zustandsraum Y liegen die *Sequenz* $S;T$, die *Auswahl* $S \square T$, und die *bedingte Auswahl* $S \boxtimes T$ jeweils in $\mathcal{G}(X \cup Y)$.
- Für $S \in \mathcal{G}(X)$ und $\mathcal{P} \in \mathbb{F}$ mit $fr(\mathcal{P}) \subseteq X$ liegt $\mathcal{P} \rightarrow S$ in $\mathcal{G}(X)$. Dabei wird $\mathcal{P}$ *guard* oder *Vorbedingung* genannt.
- Für $\mathcal{P} \in \mathbb{F}$ mit $fr(\mathcal{P}) \subseteq X$ liegt *Assert* $\mathcal{P}$ in $\mathcal{G}(X)$.
- Ist $y \notin X$ eine Variable und $S \in \mathcal{G}(X \cup \{y\})$, dann liegt die *Projektion* $@y \bullet S$ in $\mathcal{G}(X)$.
- Ist $S \in \mathcal{G}(X)$, dann liegt die *Rekursion* $\mu T.((S;T) \boxtimes Skip)$ in $\mathcal{G}(X)$.

- Für $\mathcal{P}re, \mathcal{P}ost \in \mathbb{F}$ mit $fr(\mathcal{P}re) \cup fr(\mathcal{P}ost) \subseteq X$ liegt die *operationale Spezifikation* $(\mathcal{P}re, \mathcal{P}ost)$ in $\mathcal{G}(X)$.
- Für eine Zustandsvariable $x \in X$ und Zustandsvariablen $p_1, \ldots, p_n \in X_{in}$ liegt für $S \in \mathcal{G}(X)$ auch die *Instantiierung* $x \leftarrow S[p_1, \ldots, p_n]$ in $\mathcal{G}(X \cup X_{in})$.

□

Die Definition von *Operationen* und damit von dynamischem Verhalten wird in der Spezifikationssprache durch

`Operation` <operation head> <operation body> `End` ,

mit einem Rumpf bestehend aus *guarded commands* und einem Kopf in der Form

[<output decl> ←] <command name>‘(’{ <input decl> | ‘,’}* ‘)’

beschrieben.

Beispiel 4 Sei $X = \{x\}$ ein Zustandsraum, mit x definiert wie in Beispiel 2. Dann kann die folgenden Operation definiert werden:

`Operation` setze_x $(value \in$ `byte`$) =$
  $x := value$
`End`

□

3.3 Semantik der statischen Spezifikation

Die statische Spezifikation dient im Wesentlichen dazu, Typen, also Sorten in der prädikatenlogischen Sprache $\mathcal{L}$, einzuführen. Dies bedeutet, jede in der Spezifikation definierte Menge wird als Sorte in in die Sprache aufgenommen. Zusätzlich werden für diese neue Sorte die notwendigen Prädikate, wie in Abschnitt 2.2 gefordert, definiert.

Die Definition von Variablen bzw. statischen Variablen eines Typs in der Spezifikation stellen in der prädikatenlogischen Sprache einen Variablenbezeichner der entsprechenden Sorte dar. Die Zugehörigkeit zu einer Sorte wird aus dem Kontext mittels Typisierung geschlossen.

Die Definition von Konstanten bedarf keiner weiteren Betrachtung, da diese in einer vollständigen Spezifikation nach einer entsprechenden Textersetzung überflüssig werden.

3.4 Semantik der dynamischen Spezifikation

Die Semantik der *guarded commands* aus Abschnitt 3.2 kann mittels der *Prädikattransformer* eindeutig bestimmt werden. Dazu werden im Folgenden die Abbildungen $wlp(S)$ und $wp(S)$ für alle guarded commands aus $\mathcal{G}(X)$ definiert:

- Die Anweisung $S = Skip$ soll keinen Effekt haben, d. h. es soll $\Delta(S) = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$ gelten. Aus der Definition der Prädikattransformer ergibt sich hieraus sofort
$$wlp(S)(\varphi) \equiv wp(S)(\varphi) \equiv \varphi \quad .$$

- Die Anweisung $S = Loop$ definiert einen Zustandsübergang, der niemals terminiert, d. h. $\Delta(S) = \{(\sigma, \infty) \mid \sigma \in \Sigma\}$. Hieraus folgt unmittelbar nach Definition
$$wlp(S)(\varphi) \equiv True \text{ und}$$
$$wp(S)(\varphi) \equiv False$$

- Sei S eine Zuweisung der Form $x := t$, dann ist für jede Formel $\varphi \in \mathbb{F}$
$$wlp(S)(\varphi) \equiv wp(S)(\varphi) \equiv \{x/t\}.\varphi$$

- Für $S = \mathcal{P} \to T$ werden die zugehörigen Prädikattransformer durch
$$wlp(S)(\varphi) \equiv \mathcal{P} \Rightarrow wlp(T)(\varphi) \quad \text{und}$$
$$wp(S)(\varphi) \equiv \mathcal{P} \Rightarrow wp(T)(\varphi)$$
definiert.

- Ist $S = T_1; T_2$ eine Sequenz, so ist
$$wlp(S)(\varphi) \equiv wlp(T_1)(wlp(T_2)(\varphi)) \quad \text{und}$$
$$wp(S)(\varphi) \equiv wp(T_1)(wp(T_2)(\varphi)) \quad .$$

- Für $S = T_1 \square T_2$ werden die Prädikattransformer durch
$$wlp(S)(\varphi) \equiv wlp(T_1)(\varphi) \wedge wlp(T_2)(\varphi) \quad \text{und}$$
$$wp(S)(\varphi) \equiv wp(T_1)(\varphi) \wedge wp(T_2)(\varphi)$$
definiert.

- Das *guarded command* $S = T_1 \boxtimes T_2$ ist eine abkürzende Schreibweise für $guard \to S \square \neg guard \to S$. Dabei charakterisiert $guard\ S \equiv \neg wp(S)(false)$ genau die Anfangszustände für die S terminiert. Die Definition von $wlp(S)$ und $wp(S)$ ergeben sich dann aus dem vorhergehenden Fall.

- Für $S = Assert\ \mathcal{P}$ werden durch
$$wlp(S)(\varphi) \equiv wp(S)(\varphi) \equiv \mathcal{P} \wedge \varphi$$
die dazugehörigen Prädikattransformer definiert.

- Für $S = @x \bullet T$ werden durch

$$wlp(S)(\varphi) \equiv \forall x.wlp(S)(\varphi) \quad \text{und}$$
$$wp(S)(\varphi) \equiv \forall x.wp(S)(\varphi)$$

die dazugehörigen Prädikattransformer definiert.

- Für $S = (\mathcal{P}re, \mathcal{P}ost)$ können die folgenden Prädikattransformer definiert werden:

$$wlp(S)(\varphi) \equiv \forall x'_1, \ldots, x'_n.(\mathcal{P}re \wedge \mathcal{P}ost \Rightarrow \{x_1/x'_1, \ldots, x_n/x'_n\}.\varphi)$$
$$wp(S)(True) \equiv \mathcal{P}re \Rightarrow \exists x'_1, \ldots, x'_n.\mathcal{P}ost$$

Dabei bezeichnet x' den Wert der Variablen x nach Ausführung der spezifizierten Anweisungen.
Alternativ kann eine operationale Spezifikation immer in das folgende äquivalente *guarded command* umgeformt werden:

$$\psi(S) = \mathcal{P}re \rightarrow (@x'_1, \ldots, x'_n \bullet \mathcal{P}ost \rightarrow x_1 := x'_1; \ldots; x_n := x'_n \boxtimes Loop) \quad .$$

Die Prädikattransformer ergeben sich dann entsprechend.

Beispiel 5 Für die Operation aus Beispiel 4 ist $T = (\mathcal{P}re, \mathcal{P}ost)$ mit $\mathcal{P}re \equiv True$ und $\mathcal{P}ost \equiv x' = value$ eine gültige operationale Spezifikation. Diese ist äquivalent zu folgenden *guarded command* :

$$\begin{aligned}\psi(T) &= \mathcal{P}re \rightarrow (@x' \in \texttt{byte} \bullet \mathcal{P}ost \rightarrow x := x' \boxtimes Loop)\\ &\equiv True \rightarrow (@x' \in \texttt{byte} \bullet x' = value \rightarrow x := x' \boxtimes Loop)\\ &\Leftrightarrow \underbrace{@x' \in \texttt{byte} \bullet x' = value \rightarrow x := x'}_{=:S} \boxtimes Loop\end{aligned}$$

Für das Programmstück S kann dann der Prädikattransformer $wp(S)$ berechnet werden:

$$\begin{aligned}wp(S)(\varphi) &\equiv \forall x' \in \texttt{byte}.wp(x' = value \rightarrow x := x')(\varphi)\\ &\equiv \forall x' \in \texttt{byte}.x' = value \Rightarrow wp(x := x')(\varphi)\\ &\equiv \forall x' \in \texttt{byte}.x' = value \Rightarrow \{x/x'\}.\varphi\end{aligned}$$

□

- Für $S = out \leftarrow T[p_1, \ldots, p_n]$ ergeben sich die Prädikattransformer wie folgt:

$$wlp(S)(\varphi) \equiv (\{\iota'_1/p_1, \ldots, \iota'_n/p_n\}.(wlp(T')(\{out/o'\}.\varphi)))$$
$$wp(S)(\varphi) \equiv (\{\iota'_1/p_1, \ldots, \iota'_n/p_n\}.(wp(T')(\{out/o'\}.\varphi)))$$

Dabei entsteht T' aus dem Rumpf der Operation T durch Ersetzen von out durch o' und p_i durch ι'_i $(1 \leq i \leq n)$ mit frischen Variablen o' und ι'_i $(1 \leq i \leq n)$.

Beispiel 6 Es seien die beiden folgenden Operationen definiert:

```
Operation out ∈ bool ← ist_größer (value ∈ byte) =
    (value ≥ x) → out := True□(x < 0) → out := False
End

Operation main () =
    @y ∈ bool • y ← ist_größer(7)
End
```

Dann gilt für den Rumpf T' der Operation ist_größer mit den definierten Ersetzungen zunächst

$$T' \equiv (\iota' \geq x) \rightarrow o' := True \square (\iota' < x) \rightarrow o' := False \quad .$$

Hiermit können die Prädikattransformer für den Rumpf

$$S \equiv @y \in \texttt{bool} \bullet y \leftarrow \text{ist\_größer}(7)$$

der Operation $main()$ berechnet werden. Es gilt:

$$\begin{aligned}
wlp(S)(\varphi) &\equiv \forall y \in \texttt{bool}.wlp(y \leftarrow \text{ist_größer}(7))(\varphi) \\
&\equiv \forall y \in \texttt{bool}.(\{\iota'/7\}.(wlp(T')(\{out/o'\}.\varphi))) \\
&\equiv \forall y \in \texttt{bool}.(\{\iota'/7\}.(wlp((\iota' \geq x) \rightarrow o' := True)(\{out/o'\}.\varphi) \\
&\qquad \wedge wlp((\iota' < x) \rightarrow o' := False)(\{out/o'\}.\varphi))) \\
&\equiv \forall y \in \texttt{bool}.(\{\iota'/7\}.((\iota' \geq x) \Rightarrow wlp(o' := True)(\{out/o'\}.\varphi) \\
&\qquad \wedge (\iota' < x) \Rightarrow wlp(o' := False)(\{out/o'\}.\varphi))) \\
&\equiv \forall y \in \texttt{bool}.(\{\iota'/7\}.((\iota' \geq x) \Rightarrow \{o'/True, out/o'\}.\varphi) \\
&\qquad \wedge (\iota' < x) \Rightarrow \{o'/False, out/o'\}.\varphi))) \\
&\equiv \forall y \in \texttt{bool}.((7 \geq x) \Rightarrow \{o'/True, out/o'\}.\varphi \\
&\qquad \wedge (7 < x) \Rightarrow \{o'/False, out/o'\}.\varphi)
\end{aligned}$$

□

4 Formale Beschreibung von Java

In dem vorangegangenen Abschnitt wurde eine formale Spezifikationssprache definiert, die nun verwendet werden soll, um wesentliche Bestandteile der Programmiersprache Java[10] formal zu beschreiben.

Diese Beschreibung ermöglicht es dann, wünschenswerte Eigenschaften von Java-Programmen nachzuweisen.

4.1 Einfache Datentypen

Das Ziel dieses Abschnitts ist die Beschreibung der *einfachen Datentypen* der Programmiersprache Java. Dazu werden zunächst Mengengleichungen über die von der Spezifikationssprache zur Verfügung gestellten Basismengen $NAT \subseteq INT \subseteq FLOAT$ verwendet. Bedingt durch die technische Realisierung verfügen die Java-Datentypen `byte, short, int, long, float` und `double` über einen eingeschränkten Wertebereich. Dieser kann in der Spezifikationssprache mittels Konstanten ausgedrückt werden:

$$\begin{aligned}\texttt{Constant}\ & maxbyte \in NAT = 2^7 - 1, \quad minbyte \in INT = -2^7,\\ & maxshort \in NAT = 2^{15} - 1, minshort \in INT = -2^{15},\\ & maxint \in NAT = 2^{31} - 1, \quad minint \in INT = -2^{31},\\ & maxlong \in NAT = 2^{63} - 1, \quad minlong \in INT = -2^{63}\end{aligned}$$

Auf die Darstellung der Datentypen `float` und `double` soll Aufgrund deren Komplexität hier verzichtet werden. Für die anderen einfachen Java-Datentypen ergibt sich die folgende Definition in der Spezifikationssprache:

$$\begin{aligned}\texttt{Set byte} &= \{I \in INT \mid minbyte \leq I \leq maxbyte\},\\ \texttt{short} &= \{I \in INT \mid minshort \leq I \leq maxshort\},\\ \texttt{int} &= \{I \in INT \mid minint \leq I \leq maxint\},\\ \texttt{long} &= \{I \in INT \mid minlong \leq I \leq maxlong\}\end{aligned}$$

Für die Wahrheitswerte und den Datentyp `char` können die vordefinierten Mengen $BOOL$ und $CHAR$ verwendet werden. Es gilt:

$$\begin{aligned}&\texttt{Set boolean} = BOOL,\\ &\texttt{Set char} = CHAR\end{aligned}$$

Eine Variable eines einfachen Datentyps wird in Java durch eine Definition vereinbart. Dabei legt eine Definition den Namen und die Art einer Variablen fest und sorgt gleichzeitig für die Reservierung des Speicherplatzes.

In Java gibt es drei verschiedene Arten von Variablen:

- Klassenvariablen,
- Instanzvariablen und
- lokale Variablen.

Klassenvariablen werden für jede Klasse einmal angelegt. Änderungen der Klassenvariable können von allen Instanzen einer Klasse durchgeführt werden, und haben den selben Effekt für alle Instanzen. Bei Klassenvariablen handelt es sich demnach um globale Variablen. Die Definition von Klassenvariablen in Java der Form

$$\texttt{static}\ datentyp\ name_1, \ldots, name_n; \quad (n \in \mathbb{N} \setminus \{0\})$$

wird in der Spezifikation durch Zustandsvariablen ausgedrückt:

$$\begin{array}{c} \texttt{Variable}\ name_1 \in datentyp \\ \vdots \\ \texttt{Variable}\ name_n \in datentyp \end{array}$$

Voraussetzung ist, dass für den Datentyp eine entsprechend definierte Menge in der Spezifikation vorhanden ist.

Instanzvariablen gibt es für jede angelegte Instanz einer Klasse, also für jedes Objekt. Für die Spezifikation von Instanzvariablen wird zunächst eine Möglichkeit zur Unterscheidung einzelner Objekte benötigt. Dabei ist zu beachten, dass jedes Objekt über einen vom System vergebenen eindeutigen *Objektidentifikator* $id \in ID$ verfügt. Daher ist es nahe liegend, Instanzvariablen einer Klasse als Abbildung von ID in den entsprechenden Datentyp zu definieren [11]. Die Definition von Instanzvariablen in Java der Form

$$datentyp\, name_1, \ldots, name_n; \quad (n \in \mathbb{N} \setminus \{0\})$$

wird in der Spezifikation durch Zustandsvariablen ausgedrückt:

$$\begin{array}{c} \texttt{Variable}\, name_1 \in (ID \mapsto datentyp) \\ \vdots \\ \texttt{Variable}\, name_n \in (ID \mapsto datentyp) \end{array}$$

Dabei bezeichnet $(ID \mapsto datentyp)$ die Abbildung aus der Menge der Objektidentifikatoren in die Menge der Werte des entsprechenden Datentyps.

Lokale Variablen werden in Methoden definiert. Die Gültigkeit der lokalen Variablen kann sich auf den Methodenrumpf oder auf einen inneren Block erstrecken. Dieselbe Semantik ist für das *guarded command* Projektion festgelegt, so dass die Definition von lokalen Variablen in Java der Form

$$datentyp\, name_1, \ldots, name_n; \quad (n \in \mathbb{N} \setminus \{0\})$$

in der Spezifikation wie folgt ausgedrückt werden kann:

$$\begin{array}{c} @name_1 \in datentyp\bullet \\ \vdots \\ @name_n \in datentyp\bullet \end{array}$$

Eine bei der Definition einer Variablen in Java mögliche Initialisierung wird in der Spezifikation jeweils durch eine entsprechende Zuweisung repräsentiert.

Symbolische Konstanten werden in Java durch Variablen dargestellt, die nach ihrer Initialisierung nicht verändert werden dürfen. Zur Definition von symbolischen Konstanten wird in Java das Schlüsselwort `final` verwendet. In der Spezifikation werden

konstante Klassen- bzw. Instanzvariablen wie beschrieben ausgedrückt. Die Schlüsselworte `StaticVariable` und `Variable` werden aber durch `Constant` ersetzt. Für lokale Variablen ändert sich zunächst nichts, da durch den Java-Compiler sicher gestellt wird, dass eine lokale Konstante nicht mehr geändert wird.

Es ist zu beachten, dass Konstanten durch entsprechende Textersetzungen überflüssig werden.

4.2 Referenztypen

Neben den im vorigen Abschnitt dargestellten Grundtypen sind in Java Referenzen auf Objekte wichtig. Diese werden zum Zugriff auf Instanzen einer Klasse verwendet. Sie speichern einen Identifikator, der als Adresse für ein konkretes Objekt dient.

Eine Referenz beinhaltet entweder den Identifikator eines Objekts oder den Wert `null`. Java bietet im Unterschied zu maschinennäheren Sprachen kaum Möglichkeiten, Referenzen zu manipulieren. Java beschränkt sich auf zwei Operationen: *Zuweisung* und *Dereferenzierung*.

Im ersten Fall handelt es sich um Anweisungen der Form `x = y` oder `y = null` wobei `x` und `y` Referenzen sein müssen, deren Typen zuweisungskompatibel sind.

Die Dereferenzierung wird zum Zugriff auf Attribute und Methoden eines Objekts verwendet. Dies hat in Java die Form $referenz.attribut$.

Mit der Ausnahme des `null`-Werts können Objektidentifikatoren nicht direkt angegeben werden. Sie werden bei der Konstruktion eines Objekt erzeugt. Der `new`-Operator erstellt das Objekt im freien Speicher und ruft für dieses den Konstruktor der Klasse auf.

Identifikatoren sind eindeutig, d.h. keine zwei Objekte besitzen den gleichen Identifikator. Diese Eigenschaft ist Teil der Semantik des `new`-Operators. Eine Implementierung der hier vorgestellten Spezifikationssprache kann dieses Verhalten realisieren durch eine Integer-Darstellung der *ID*s in der Form

$$\texttt{Variable}\, lastnew \in \texttt{int} = 0$$

$$lastnew := lastnew + 1;$$
$$@obj \in \texttt{int}\bullet$$
$$\quad obj := lastnew;$$
$$\quad \ldots$$

Denkbar ist natürlich jede Darstellung, die die obige Eindeutigkeitseigenschaft garantiert.

Die Sprache der *guarded commands* beinhaltet unabhängig von der konkreten Realisierung den Typ *ID*, der für beliebige Referenzen stehen kann. Die Klassenstruktur (siehe auch Abschnitt 4.7) wird durch die Übersetzung in die guarded commands irrelevant; ein einziger Typ ist ausreichend.

Die Konstruktion von Identifikatoren wird, wie schon dargestellt, über die Schlüsselwörter `null` und `new` realisiert. Dabei steht `null` für eine Nullreferenz analog zu Java und `new` für den neuen Objektidentifikator.

Beispiel 7 Die Konstruktion eines Objekts in Java in

```
class Beispiel {
    int x;
}

Beispiel b = new Beispiel();
b.x = 4;
```

kann mit dieses Mitteln in die *guarded commands* übersetzt werden. Es ergibt sich

$$
\begin{aligned}
&@b \in \mathit{ID} \bullet \\
&\quad b := \texttt{new}; \\
&\quad \mathit{Beispiel\_Beispiel}(b); \\
&\quad x[b] := 4
\end{aligned}
$$

Dabei steht $x[b]$ für die Anwendung der Abbildung $x : \mathit{ID} \mapsto \mathit{Integer}$ auf die Referenz b.

4.3 Felder

Java behandelt Felder als Objekte mit impliziten, erweiterten Eigenschaften. Jede Feldvariable besitzt ein Attribut `length`, das die Größe des Feldes angibt. Über den `[]`-Operator kann auf die Feldelemente 0 bis $\mathit{length} - 1$ zugegriffen werden. Das folgende Beispiel zeigt die Erstellung eines Feldes gefolgt von einem Zugriff über diesen Operator:

Beispiel 8
```
int[] x = new int[10];
x[4] = 42;
```

Bei der Übersetzung in die *guarded commands* wird eine Objektvariable für die Länge der Felder angelegt. Diese existiert nur einmal für alle Feldtypen:

$$\texttt{Variable}\ \mathit{array\_length} \in (\mathit{ID} \mapsto \texttt{int})$$

Die Darstellung der Feldelemente erfolgt analog zur Umwandlung von Objektvariablen. Der Hauptunterschied besteht darin, dass für den Zugriff auf ein Feldelement die Referenz auf das Feldobjekt und zusätzlich der Feldindex benötigt wird. In den *guarded commands* erzwingt dies eine zweidimensionale Abbildung:

$$\texttt{Variable}\ \mathit{array\_int\_elements} \in (\mathit{ID} \mapsto \texttt{int} \mapsto \texttt{int})$$

Der Zugriff über den `[]`-Operator bekommt in den *guarded command* die Form

$$\texttt{array\_int\_elements}[x][4]$$

Wichtig dabei ist, dass für alle einfachen Datentypen (siehe 4.1) eine eigene Elementvariable angelegt wird, um eine korrekte Typisierung zu ermöglichen. So ergibt die Instanzierung eines Feldes aus `boolean`-Werten die Elementvariable für `boolean` in den guarded commands:

$$\texttt{Variable}\, array\_boolean\_elements \in (ID \mapsto \texttt{int} \mapsto \texttt{boolean})$$

Für Felder aus Objektreferenzen ist dagegen nur eine solche Elementvariable notwendig. Die Unterscheidung zwischen verschiedenen Referenztypen wird durch die Übersetzung in *guarded commands* aufgehoben:

$$\texttt{Variable}\, array\_ID\_elements \in (ID \mapsto \texttt{int} \mapsto ID)$$

Die Konstruktion einer Feldvariable wird analog zur normalen Objektkonstruktion durchgeführt (siehe 4.2). Aus Beispiel 8 ergibt sich dabei die *guarded command* Darstellung

$$\begin{aligned}
&@x \in ID \bullet \\
&\quad x := \texttt{new}; \\
&\quad array_length[x] := 10; \\
&\quad array_int_elements[x][4] := 42;
\end{aligned}$$

4.4 Ausdrücke

Die Bedeutung von Java-Ausdrücken ist in weiten Teilen sehr einfach und kann direkt in die *guarded command* Darstellung übernommen werden. Die Java-Operatoren ohne Seiteneffekte werden auch von der Spezifikationssprache zur Verfügung gestellt. Dabei wird Arithmetik wie gewohnt mit endlicher Genauigkeit und Überlauf durchgeführt.

Eine besondere Beachtung bedürfen jedoch Funktionsaufrufe und Kurzschreibweisen wie `i++`, da sie Seiteneffekte verursachen. Ausdrücke der *guarded commands* sind grundsätzlich statisch. Ihre Anwendung verändert keine Variablen. Alle Java-Ausdrücke, die dieser Regel widersprechen, sind entsprechend umzuformen.

Aus dieser Forderung folgt, dass Funktionsaufrufe innerhalb von Termen herausgezogen werden müssen. Dies ist durch das Anlegen einer temporären Variable problemlos möglich.

Beispiel 9 Aus dem Java-Programmstück

```
x := beispiel( 5 ) + 1;
```

wird die *guarded command* Darstellung

$$
\begin{aligned}
&@temp \in \texttt{int}\bullet \\
&\quad temp \leftarrow beispiel(5); \\
&\quad x := temp + 1
\end{aligned}
$$

Die Inkrement- und Dekrement-Operatoren `++` bzw. `--` können auf ähnliche Weise transformiert werden. Dabei ist zu unterscheiden zwischen den Prefixoperatoren, bei denen der veränderte Wert der Variable geliefert wird, und den Postfixoperatoren, bei denen der ursprüngliche Wert geliefert wird. In jedem Fall wird eine temporäre Variable benötigt, die den Wert aufnimmt, der im ursprünglichen Term gewirkt hat.

Beispiel 10 Aus dem Java-Programmstück

```
x := (i++) + 1;
y := (++j) + 2;
```

wird die *guarded command* Darstellung

$$
\begin{aligned}
&@temp \in \texttt{int}\bullet \\
&\quad temp := i; \\
&\quad i := i + 1; \\
&\quad x := temp + 1; \\
&@temp \in \texttt{int}\bullet \\
&\quad j := j + 1; \\
&\quad temp := j; \\
&\quad y := temp + 2
\end{aligned}
$$

Analoge Umformungen ergeben sich für die beiden Dekrement-Operatoren.

Tauchen in einem Term mehrere dieser Seiteneffekte auf, so sind entsprechend viele temporäre Variablen anzulegen, um das sequentielle Auswerten der Terme durch die Java-Virtual-Machine explizit zu realisieren.

Bei der Verwendung der booleschen Operatoren `||` und `&&` ist eine weitere komplexere Transformation notwendig. Beim `||`-Operator wird bei einem Wahrheitswert `true` auf der linken Seite die rechte Seite gar nicht mehr ausgewertet, was bei der Umformung von Funktionaufrufen und Inkrement-/Dekrement-Operatoren beachtet werden muss. Das Gleiche geschieht bei `false` auf der linken Seite von `&&`. In beiden Fällen müssen die explizit ausgeführten Seiteneffekte durch *guards* in ihrer Ausführung beschränkt werden.

4.5 Anweisungen

In diesem Abschnitt wird die Übersetzung der wesentlichen Java-Anweisungen in die definierte zustandsbasierte Spezifikationssprache beschrieben. Dabei werden die Java-Anweisungen durch *guarded commands* ausgedrückt.

Es ist zunächst zu beachten, dass eine Folge von Java-Anweisungen, die im Quelltext durch Semikola getrennt sind, zu einer Sequenz von *guarded commands* führt. Die Definition einer lokalen Variablen in Java kann wie beschrieben auf eine Projektion in der Spezifikationssprache abgebildet werden. Dabei ist zu beachten, dass lokale Variablen in Java nicht mit einem Standardwert initialisiert werden. Wird eine lokale Variable in Java mit einem Startwert initialisiert, ergibt dies in der Spezifikation eine Zuweisung als erstes *guarded command* nach der Projektion. Die Beschreibung der Übersetzung der komplexeren Java-Anweisungen erfolgt nun Zug um Zug.

- Die Anweisung `assert <Expr> [ : <Expr> ]` ergibt in der Spezifikation stets das *guarded command*

  $$\text{Assert } \mathcal{P}$$

  mit der seiteneffektfreien Darstellung $\mathcal{P}$ des ersten Ausdrucks. Der zweite Ausdruck ist für die weitere Betrachtung unerheblich, da er im Allgemeinen einen Text darstellt, der dem Anwender im Fehlerfall angezeigt wird. Daher kann der zweite Ausdruck in der Übersetzung vernachlässigt werden.

- Eine bedingte Anweisung

  ```
  if (<Expression>) <Stmt1> [else <Stmt2>]
  ```

  wird durch

  $$(\text{<Expr>}) \rightarrow \text{<Stmt\_1>} \boxtimes \text{<Stmt\_2>}$$

  in *guarded commands* ausgedrückt. Dabei ist zu beachten, dass bei einer `if`-Anweisung ohne `else`-Zweig für <Stmt_2> das *guarded command* $Skip$ eingesetzt wird.

  Beispiel 11 Das Javaprogrammstück

  ```
  if (x > y)
          x -= y;
  else
          y -= x;
  ```

  ergibt die folgenden *guarded commands* :

  $$(x > y) \rightarrow x := x - y \boxtimes y := y - x \quad .$$

- Eine `switch`-Anweisung der Form

  ```
  switch (<Expr>) {
     case <Expr_0> : <Stmt_0>
                     break;
  ```

```
   case <Expr_n> : <Stmt_n>
                   break;
   default:        <Stmt>
}
```

führen wie die `if`-Anweisungen zu einem *guarded command*

$$\ldots \boxtimes (\text{<Expr\_i>=<Expr>}) \rightarrow \text{<Stmt\_i>} \boxtimes \ldots$$

Zu beachten ist, dass, wenn ein `case`-Rumpf nicht mit einer `break`-Anweisung abgeschlossen ist, auch die Rümpfe der folgenden `case`-Anweisungen bis zum ersten `break` ausgeführt werden. Dies gilt insbesondere auch für die `default`-Anweisung, die an einer beliebigen Stelle in der `switch`-Anweisung auftreten kann.

- Schleifen der Form `while (<Expr>) <Stmt>` ergeben

$$\mu T.((\text{<Expr>} \rightarrow \text{<Stmt>}; T) \boxtimes Skip)$$

als *guarded command* mit einer Variablen T als Repräsentant eines *guarded commands* .

Beispiel 12 Die Schleife

```
while (x != y)
   if (x > y)
        x -= y;
   else
        y -= x;
```

mit dem Schleifenrumpf aus Beispiel 12 wird durch

$$\mu T.((\neg(x = y) \rightarrow (x > y) \rightarrow (x := x - y \boxtimes y := y - x); T) \boxtimes Skip)$$

repräsentiert.

Tritt im Schleifenrumpf eine `break`-Anweisung auf, so bedarf es einer gesonderten Behandlung der Schleife. Auf die hier aber nicht näher eingegangen werden soll.

- Eine Schleife der Form `do <Stmt> while (<Expr>)` ist immer äquivalent zu `<Stmt>; while (<Expr>) <Stmt>`. Daraus ergibt sich die Übersetzung in *guarded commands* wie für `while`-Anweisungen beschrieben.

Beispiel 13 Die Schleife

```
do {
   x += 2;
   if (x==100) {
        y -= x;
   }
   else
        x -= 3;
} while (x != 0)
```

kann wie folgt in *guarded commands* dargestellt werden:

$x := x + 2; (x = 100) \rightarrow y := y - x \boxtimes x := x - 3;$
$\mu T.((x := x + 2; (x = 100) \rightarrow y := y - x \boxtimes x := x - 3; T) \boxtimes Skip)$

- Eine `for`-Schleife

  ```
  for (<Stmt_1>; <Expr>; <Stmt_2> ) <Stmt_3>
  ```

  ist äquivalent zu einer `while`-Schleife der Form

  ```
  <Stmt_1>; while (<Expr>) { <Stmt_3>; <Stmt_2>}
  ```

  Daraus ergibt sich die Übersetzung in die *guarded commands* wie für `while`-Anweisungen beschrieben.

4.6 Methoden

In diesem Abschnitt soll die Übersetzung von Methoden in die guarded command Operationen beschrieben werden.

Java unterscheidet analog zu den Variablen zwischen statischen und nichtstatischen Methoden. Erstere erlauben eine fast triviale Übersetzung. Nachdem alle Verweise auf Klassenvariablen aufgelöst sind, können diese eins zu eins umgeformt werden.

Beispiel 14 Aus dem Java-Beispiel

```
class Beispiel {
  static int x;

  static int add( int d ) {
    x = x + d;
    return x;
  }
}
```

kann direkt die *guarded command* Variante

`Variable` $Beispiel\_x \in int$

`Operation` $out \in$ `int` $\leftarrow$ Beispiel_add $(d \in$ `int`$) =$
    $Beispiel\_x := Beispiel\_x + d;$
    $out := Beispiel\_x;$
`End`

erzeugt werden.

Bei der Umwandlung von nichtstatischen Methoden sind ähnliche Überlegungen wie bei Objektvariablen notwendig. Nichtstatische Methoden können auf Variablen des aktuellen Objekts zugreifen. Dazu benötigen sie den Identifikator der Objektinstanz. Java speichert diesen in der Variable `this`. Diese wird in Java nicht explizit definiert, sondern wird als impliziter Parameter von nichtstatischen Methoden beim Aufruf übergeben. Die Methode in

```
class Beispiel {
  int x;

  int add( int d ) {
    x = x + d;
    return x;
  }
}
```

wird von Java intern (d.h. vom Java-Compiler und der Java-Virtual-Machine) so behandelt, als wäre sie in der folgenden expliziten Form definiert:

```
int add( Beispiel this, int d ) {
    this.x = this.x + d;
    return this.x;
}
```

Bei der Übersetzung in die *guarded commands* wird dieser Schritt explizit ausgeführt, so dass

`Variable` $Beispiel\_x \in (ID \mapsto int)$

`Operation` $out \in$ `int` $\leftarrow$ Beispiel_add $(this \in ID, d \in$ `int`$) =$
    $Beispiel\_x[this] := Beispiel\_x[this] + d;$
    $out := Beispiel\_x[this];$
`End`

als resultierende Darstellung entsteht.

Die Semantik von Methoden, deren Rumpf nicht vorliegt, muss mittels einer operationalen Spezifikation ausgedrückt werden. An die Stelle des Methodenaufrufs tritt in der Spezifikation dann ein Formelpaar $(\mathcal{P}re, \mathcal{P}ost)$.

In Java wird keine eindeutige Bezeichnung von Methoden gefordert. Für den Fall von Mehrdeutigkeiten wird in der Java Language Specification [12] ein Algorithmus beschrieben, der anhand der Datentypen der Parameter und der Anzahl derselben eine der in Frage kommenden Methoden auswählt. Diese Auswahl findet bei Java während der Übersetzung durch den Compiler statt. Bei der Transformation in die *guarded commands* müssen gleichbenannten Java-Methoden mit unterschiedlichen Signaturen (d.h. mit unterschiedlichen Parametertypen bzw. unterschiedlicher Parameteranzahl) unterscheidbare Bezeichner zugewiesen werden. Die implizite Semantik, die von dem oben erwähnten Algorithmus vorgegeben wird, wird so in den *guarded commands* durch eine eindeutige Methodenbezeichnung ersetzt.

Offen bleibt die Frage, wie Polymorphie behandelt wird. Neben Mehrdeutigkeiten während der Kompilierung erlaubt Java zusätzlich das so genannte Überschreiben von Methoden. Eine abgeleitete Klasse kann die Implementierung einer Signatur durch eine eigene ersetzen. Welche dieser Implementierungen während der Laufzeit aufgerufen wird, hängt vom Typ der betrachteten Referenz ab und ist somit rein dynamisch. Die hier dargestellte Spezifikationssprache und deren formale Semantik geht auf die Behandlung von Polymorphie noch nicht ein. Hier ist weitere Forschung notwendig.

4.7 Vererbung

Die Übersetzung von Java-Klassen in die *guarded commands* beseitigt die Klassenstruktur, d.h. die Zugehörigkeit der Attribute und Methoden sowie die Typisierung von Referenzen wird ignoriert. Während diese Informationen für die Semantik von Programmen im Allgemeinen unwichtig sind (dies ermöglicht erst die beschriebene Übersetzung), erlaubt es Java, auf die Klassenstruktur explizit zu verweisen. Dies geschieht über den `instanceof`-Operator, der den Wahrheitswert `true` genau dann liefert, wenn die Objektreferenz auf der linken Seite auf ein Objekt zeigt, welches die Klasse auf der rechten Seite als Elternklasse besitzt. Eine Anwendung in Java sieht z.B. so aus:

```
if( x instanceof java.util.Vector ) {
  java.util.Vector v = (java.util.Vector)x;
}
```

Nun sind die Klasseninformation durch die Transformation in die *guarded commands* nicht mehr vorhanden. Jedoch können diese in einem Feld `type` abgelegt werden:

`Variable` $type \in (ID \mapsto int)$

Für die Speicherung der Typinformation sind verschiedene Darstellungen denkbar. Eine Klassenstruktur hat eine Baumstruktur (wenn von Interfaces abgesehen wird) mit `java.lang.Object` als Wurzel. Eine naheliegende Darstellung kann durch eine

Bitdarstellung als `int` realisiert werden. Jeder Klasse wird ein Bit zugeordnet. Die `instanceof`-Operation kann dann in einen einfachen Bittest überführt werden.

Nach der Konstruktion eines Objekts durch den `new`-Operator muss das `type`-Feld initialisiert werden. Für das Beispiel 7 ergibt dies die folgenden *guarded commands* :

$$
\begin{aligned}
&@b \in \mathit{ID} \bullet \\
&\quad b := \texttt{new}; \\
&\quad \mathit{type}[b] := 1 + 2; \\
&\quad \mathit{Beispiel\_Beispiel}(b); \\
&\quad x[b] := 4
\end{aligned}
$$

Dabei wird davon ausgegangen, dass das niederwertigste Bit der Klasse `java.lang.Object` und das Bit mit der Wertigkeit 2 der `Beispiel`-Klasse zugeordnet wird.

Ein Test auf die Zuhörigkeit zur Klasse `Beispiel` bekommt damit die Form:

$$(\mathit{type}[b] \,\&\, 2 == 2) \rightarrow \ldots$$

Die Semantik von Java-Programmen wird auch von Ausnahmen (Exceptions) geprägt, die die Java-Virtual-Machine während der Ausführung werfen kann. Diese werden in dieser Spezifikationssprache noch nicht modelliert. Bei expliziten Typumwandlungen werden bei inkompatiblen Typen `ClassCastExcpetions` geworfen. Die Information in `type` könnten auch für die Modellierung dieses Verhaltens verwendet werden.

5 Beispiel

Anhand einer Java-Implementierung des Euklidischen-Algorithmus soll in diesem Abschnitt die Übersetzung von Java-Programmen in die beschriebene Spezifikationssprache umfassender betrachtet werden.

Der folgende Java-Quellcode implementiert den Euklidischen-Algorithmus:

```
public class Euklid {
   int ggT;

   Euklid() {
      ggT = 0;
   }

   int ggtIterativ( int x, int y ) {
      while( x != y )
         if( x > y )
```

```
            x -= y;
         else
            y -= x;
      return x;
   }

   public static void main( String[] args ) {
      Euklid e = new Euklid();

      int x = 382;
      int y = 576;

      e.ggT = e.ggtIterativ( x, y );
   }
}
```

Die beschriebene Übersetzung in die definierten *guarded commands* ergibt dann die folgende Darstellung in der Spezifikationssprache. Dabei ist insbesondere zu beachten, dass die Klassenstruktur nicht mehr benötigt wird.

`Set` $Integer = \{I \in INT \mid -2^{31} \leq I \leq 2^{31} - 1\}$

`Variable` m_Euklid_ggT $\in (ID \mapsto Integer)$

`Operation` o_out_2 ← op_Euklid_ggtIterativ(this, i_x, i_y) =
    μ *_loop* . (¬ (i_x = i_y)) →
        ((i_y < i_x)) → i_x := (i_x - i_y)
        End ⊠ i_y := (i_y - i_x); _loop
    End ⊠ Skip;
    o_out_2 := i_x
End

`Operation` stop_Euklid_main(i_args) =
    @l_e ∈ *ID*• l_e ← new;
    op_Euklid_Euklid(l_e);
    @l_x ∈ $Integer$• l_x := 382;
    @l_y ∈ $Integer$• l_y := 576;
    m_Euklid_ggT[l_e] ← op_Euklid_ggtIterativ(l_e, l_x, l_y)
`End`

`Operation` op_Euklid_Euklid(this) =
    op_Object_Object(this);
    m_Euklid_ggT[this] := 0;
`End`

```
Operation op_Object_Object(this) =
End
```

6 Zusammenfassung

Der vorliegende Bericht beschreibt die Übersetzung von Java-Programmen in eine auf den *guarded commands* aufbauende Spezifikationssprache. Die Schwierigkeit liegt in der Notwendigkeit die durch die Java-Virtual-Machine implizit gegebene Semantik von Java-Konstrukten explizit ausdrücken zu müssen. Insbesondere dürfen in der Spezifikation keine Ausdrücke mit Seiteneffekten auftreten.

Durch die Übersetzung wird mittels der für die Spezifikationssprache definierten Semantik nach Dijkstra eine Semantik in Java-Programme induziert. Diese kann dazu verwendet werden, wünschenswerte Eigenschaften von Java-Programmmen nachzuweisen. Es ist insbesondere möglich, typische Laufzeitfehler schon am Quelltext zu erkennen.

Literatur

[1] Clark, E.: Language Constructs for which it is impossible to obtain good Hoare-like axioms. *Journal of the ACM* (1979) 26(1):129–147

[2] Alves-Foss, J.: *Formal Syntax and Semantics of Java.* Springer, LNCS 1523, Berlin Heidelberg New York (1999).

[3] Leino, K.R.M.: Extended Static Checking: a Ten-Year Perspective. *Lecture Notes on Artificial Intelligence* (2001) LNCS 2000:157–175

[4] Leino, K.R.M., Saxe, J., Stata, R. : Checking Java Programms via guarded commands. *Technical Note 1999-002 Compaq Systems Research Center* (1999).

[5] Schewe, K. D.: Modular State-Based Specifications Emphasizing Static and Dynamic Consistency. *Technical Report 5/2000 Massey University* (2000).

[6] Bell J., Machover M. *A Course in Mathematical Logic*. North-Holland, Amsterdam London New York Tokyo (1977).

[7] Dijkstra, E. W., Scholten, C. S. *Predicate calculus and program semantics*. Springer, Berlin Heidelberg New York (1989).

[8] Dijkstra, E. W.: *A Discipline on Programming*. Prentice Hall, Englewood Cliffs, New Jersey (1976).

[9] Nelson, G. A generalization of Dijkstra's calculus. *ACM Transaction on Programming Languages and Systems* (1989) 11(4):517–561

[10] Arnold, K., Gosling, M., Holmes, D.: *The Java Programming Language, Third Edition.* Addison-Wesley, Pearson Education (2000).

[11] Leino, K. R. M. *Toward Reliable Modular Programs.* PhD Thesis, California Institute of Technology, Pasadena (1995).

[12] Gosling, J., Joy, B., Steele, G., Bracha, G. *The Java Language Specification, Second Edition* Sun Microsystems (2000).

A Formale Notation der Spezifikationssprache

Dieser Anhang beschreibt vollständig die verwendete formale Notation der Spezifikationssprache in BNF mit dem Startsymbol <specification>.

Die Metasyntax verwendet | für Alternativen, $[\cdot]$ für optionale Terme und $\{$<*exp*> | <*s*>$\}^*$ oder $\{$<*exp*> | <*s*>$\}^+$ für die Iteration von exp mit <s> als Trennsymbol. Wenn die $+$-Version verwendet wird, muss exp mindestens einmal auftreten.

Die verwendeten Schlüsselwörter sind in alphabetischer Reihenfolge:

```
Assert   Constant   End   Operation
Post     Pre        Set   Variable
```

Syntax der Mengenspezifikationen

<set specification> ::= `Set` <set name> ‘=’ <set expression>
<set name> ::= <token>
<set expression> ::= $BOOL$ | $CHAR$ | $STRING$ |
NAT | INT | $FLOAT$ | ID |
<set name> |
‘(’ <set expression> $\mapsto$ <set expression> ‘)’ |
<set comprehension>
<set comprehension> ::= ‘{’ <variable name> ‘$\in$’ <set expression> ‘|’
<formula> ‘}’
<variable name> ::= <token>

Syntax der Variablen- und Konstantenspezifikationen

<variables specification> ::= <variable specification> | <constant specification>
<variable specification> ::= `Variable` <variable name> ‘$\in$’ <set expression>
‘=’ <term>
<variable name> ::= <token>
<constant specification> ::= `Constant` <constant name> ‘$\in$’ <set expression>
‘=’ <term>
<constant name> ::= <token>

Terme und Formeln

<term> ::= **null** |
<variable> |
<constant name> |
<token> |
<string term>
<boolean term> |
<arithmetic term>
<variable> ::= <variable name> | <primed variable>
<primed variable> ::= <variable name>‘’’

<string term> ::= <string> | <varible name> |
{ <string term> | ‘$\sim$’ }$^+$
<arithmetic term> ::= <natural number> | <integer> |
<floating point number> | <variable> |
‘(’ <arithmetic term> ‘)’ |
‘$-$’ <arithmetic term> |
<arithmetic term> <arith op> <arithmetic term>
<arith op> ::= ‘$+$’ | ‘$-$’ | $\cdot$ | / | mod
<boolean term> ::= <formula>
<formula> ::= ‘$True$’ |
‘$False$’ |
‘(’ <formula> ‘)’ |
‘$\neg$’ <formula> |
<formula> ‘$\wedge$’ <formula> |
<formula> ‘$\vee$’ <formula> |
<formula> ‘$\Rightarrow$’ <formula> |
<formula> ‘$\Leftrightarrow$’ <formula> |
‘$\forall$’ <variable name> ‘$\in$’ <set expression>
‘.’ <formula> |
‘$\exists$’ <variable name> ‘$\in$’ <set expression>
‘.’ <formula> |
<term> <comparison operator> <term>
<comparison operator> ::= ‘$=$’ | ‘$\neq$’ | $>$ | $<$ | $\leq$ | $\geq$ | $\in$

Syntax der Operationen

<operation specification> ::= <operation head>
‘$=$’ <operation body>
`End`
<operation head> ::= [<output declaration> ‘$\leftarrow$’]
<command name>
[‘(’<input declaration> ‘)’]
<command name> ::= <token>
<output declaration> ::= <variable name> ‘$\in$’ <set expression>
<input declaration> ::= { <variable name> ‘$\in$’ <set expression> | ‘,’ }$^+$
<operation body> ::= <guarded command> |
<pre post specification>
<guarded command> ::= ‘$Skip$’ |
‘$Loop$’ |
<variable name> ‘:=’ <term> |
<formula> ‘$\rightarrow$’ <operation body> |
{ <operation body> | ‘;’ }$^+$ |
{ <operation body> | ‘$\square$’ }$^+$ |
{ <operation body> | ‘$\boxtimes$’ }$^+$ |

‘(’ <guarded command> ‘)’ |
‘μ’ <command name> ‘.’ <operation body> |
‘@’ <variable name> ‘$\in$’ <set expression> ‘$\bullet$’
<operation body> |
`Assert` <formula> |
[<variable name> ‘$\leftarrow$’] <command name>

<command name> ::= <token>
<pre post specification> ::= `Pre` <formula> `Post` <formula>

Spezifikation

<specification> ::= { <set specification> | }*
{ <variables specification> | }*
{ <operation specification> | }*